Limbajul de programare este o notație formală. Ea se fundamentează prin seturi de reguli care definesc forma şi înțelesul construcțiilor limbajului. Pe baza acestor reguli se stabileşte (de către programator sau de către compilator) dacă un anumit program este corect şi care este sensul lui, cu alte cuvinte ce trebuie să se întâmple atunci când el va fi executat. Regulile prin care se defineşte limbajul se referă la două aspecte majore ale acestuia: *sintaxa* şi *semantica*. Sintaxa este aceea care determină forma construcțiilor limbajului în timp ce semantica stabileşte sensul asociat construcțiilor corecte din punct de vedere sintactic.

Un limbaj de programare poate fi specificat prin tripletul:

$L=<Sm, St, f: Sm \rightarrow St>$, unde:

- Sm reprezintă semantica limbajului;
- St este sintaxa sa;
- f este funcția de asociere a semanticii, unei sintaxe date.

Este evident că o descriere formală completă a limbajului trebuie să trateze toate aceste aspecte.

În principiu, orice metodă de definire formală a limbajelor de programare oferă soluții pentru rezolvarea următoarelor probleme:

- Definirea unui alfabet de simboluri de bază A şi, implicit, definirea mulțimii A^* reprezentând toate sirurile de simboluri posibile care pot fi construite cu elementele din A.
- Furnizarea unui set de reguli pentru selectarea mulțimii $P \subseteq A^*$ a programelor corecte din punct de vedere sintactic.
- Specificarea semanticii fiecărui element $p \in P$.

Sintaxa

Regulile de sintaxă determină o mulțime infinită de **propoziții**, corecte din punct de vedere al formei lor. Desigur, doar o parte dintre acestea sunt corecte şi din punct de vedere al regulilor semantice. Elementele constituente ale propozițiilor se numesc **simboluri**. Modul în care caracterele care formează **alfabetul** limbajului, se grupează formând simboluri este descris prin regulile lexicale. Acestea pot fi considerate ca făcând parte din sintaxa limbajului. Totalitatea simbolurilor formează **vocabularul** limbajului. Din vocabular fac parte identificatori, cuvinte cheie (ca `begin`, `var`, `mod` în Pascal), operatori (`+`, `++`, `<=`, `==`, | în C), literale (întregi, flotante, şiruri de caractere).

Specificarea sintaxei prin gramatici

Totalitatea regulilor sintactice ale unui limbaj reprezintă *gramatica* limbajului. Se pune problema cum se descrie gramatica unui limbaj? O metodă foarte răspândită este notația folosită pentru prima dată la descrierea limbajului Algol 60 şi cunoscută sub numele **BNF** (Backus Naur Form). Vom prezenta o formă uşor extinsă a acestei notații, aşa numitul BNF extins.

BNF este un metalimbaj (un limbaj utilizat pentru descrierea altui limbaj) care foloseşte următoarele metasimboluri:

::=	însemnând *definit ca*
|	însemnând *sau*
< şi >	sunt folosite pentru a include neterminale
[şi]	sunt folosite pentru a include secvențe opționale
{ şi }	sunt folosite pentru a include secvențe care se pot repeta de oricâte ori (inclusiv de zero ori).

Sintaxa este definită ca o succesiune de relații (reguli) BNF. Fiecare relație defineşte un neterminal specificat în stânga semnului **::=.** În dreapta semnului găsim o succesiune de neterminale şi terminale. Terminalele sunt simboluri ale limbajului. Fiecare neterminal găsit în partea dreaptă a unei relații trebuie să fie definit într-o altă relație. Gramatica completă a limbajului reprezintă un set de relații prin care sunt definite toate neterminalele. Un anumit neterminal particular este aşa numitul *simbol de start*; acesta se numeşte în mod tipic ⟨ `program` ⟩.

Un program reprezintă o succesiune de simboluri deci, în termenii de mai sus, de terminale. Programul este corect din punct de vedere sintactic, dacă succesiunea de simboluri poate fi derivată (se poate genera) pe baza regulilor gramaticale, pornind de la simbolul de start.

Verificarea corectitudinii din punct de vedere sintactic a unui program, prin derivarea unui arbore sintactic, se numeşte *analiză sintactică*. O astfel de analiză se poate desfăşura, în principiu, în două feluri:

- *De jos în sus*: se porneşte de la succesiunea de simboluri, reprezentând programul, şi se înlocuiesc succesiv secvențe reprezentând partea dreaptă a unor reguli cu partea stângă corespunzătoare, până se ajunge la simbolul de start (dacă programul este corect).

- *De sus în jos*: pornind de la simbolul de start, şi înlocuind succesiv neterminalele conform cu regulile gramaticii: se încearcă derivarea succesiunii de simboluri corespunzătoare programului.

Expresii regulate

Unele reguli sintactice mai simple, cum sunt cele lexicale, se pot exprima prin expresii regulate. Fiecare expresie regulată e desemnează o mulțime de şiruri S formate cu elementele unui alfabet A prin aplicarea unor operatori specifici, derivați direct din principalele operații definite între mulțimile de şiruri.

Considerând S, $S1$ şi $S2$ astfel de mulțimi de şiruri, operațiile care ne interesează sunt:

a) **Reuniunea**: $S_1 \cup S_2 = \{s \mid s \in S_1 \text{ sau } s \in S_2\}$

b) **Produsul** sau **Concatenarea**: $S_1S_2 = \{s_1s_2 \mid s_1 \in S_1 \text{ şi } s_2 \in S_2\}$

c) **Puterea**, se defineşte pe baza produsului, astfel:

$$S^n = \begin{cases} \{\varepsilon\} & n=0, \text{ ε-şirul vid} \\ S^{n-1}S & \forall n \in N, n \geq 1 \end{cases}$$

d) **Închiderea Kleene** sau **Stea**:

$$S^* = \bigcup_{i=0}^{\infty} S^i$$

e) **Inchiderea pozitivă** sau **Plus**:

$$S^+ = \bigcup_{i=1}^{\infty} S^i$$

Observație: $S^* = S^+ \cup \{\varepsilon\}$

Exemple: Se consideră următoarele mulțimi de şiruri de lungime unu:

$L=\{A,B,\ldots,Z,a,b,\ldots,z\}$
$C=\{0,1,\ldots,9\}$

Aplicând unele operații dintre cele introduse anterior se pot defini următoarele mulțimi noi:

$L \cup C$	- mulțimea tuturor literelor şi cifrelor
LC	- toate şirurile de lungime doi în care primul caracter este literă iar al doilea, cifră
L^4	- mulțimea tuturor şirurilor formate din patru litere
L^*	- mulțimea tuturor şirurilor posibile de litere, de orice lungime, formate pe baza schemei aranjamentelor cu repetiție, inclusiv şirul vid
C^+	- mulțimea tuturor şirurilor de cifre care conțin cel puțin o cifră
$L(L \cup C)^*$	- mulțimea identificatorilor: toate şirurile de litere şi cifre care încep cu o literă.

O **expresie regulată** se poate construi pornind de la elementele unui alfabet A, pe baza următoarelor trei reguli:

1. ε este o expresie regulată.

2. a este o expresie regulată, $\forall a \in A$.

3. Se consideră expresiile regulate notate e_1, e_2 şi e desemnând mulțimile de şiruri S_1, S_2 şi respectiv S. Între aceste expresii se pot aplica următorii operatori, derivați direct din operațiile între şiruri prezentate anterior, rezultatul fiind tot o expresie regulată:

a) **Reuniunea** : $(e1)|(e2)$ desemnând mulțimea $S_1 \cup S_2$

b) **Produsul** sau **Concatenarea** :$(e1)(e2)$ desemnând mulțimea S_1S_2

c) **Stea** : $(e)^*$ desemnând mulțimea $(S)^*$.

Observație: (e) desemnează mulțimea de şiruri S, adică este echivalentă cu e. În consecință, dacă nu există pericolul unor ambiguități, într-o expresie regulată se poate renunța la parantezele redondante. Pentru aceasta trebuie ținut cont de următoarele proprietăți algebrice ale operatorilor:

- Toți cei trei operatori sunt asociați la stânga
- Operatorul unar stea (*) are prioritatea cea mai înnaltă. Urmează, în ordine, produsul şi reuniunea (|).

Exemplu: $litera(litera|cifra)^*$ este o expresie regulată care desemnează mulțimea identificatorilor: mulțimea $L(L \cup C)^*$ din exemplele anterioare.

Având în vedere faptul că anumite construcții apar frecvent în descrierea expresiilor regulate, cele trei operații de bază pot fi extinse cu următoarele notații suplimentare:

d) **Unul sau mai multe exemplare**

Pornind de la semnificația stabilită pentru operatorul unar postfixat **+** prin operația de închidere pozitivă, expresia `ee*` este echivalentă cu e^+.

e) **Zero sau un exemplar**

Se introduce operatorul unar postfixat `?` astfel încât expresia e|ε devine echivalentă cu `e?`.

f) **Clase de caractere**

Notația $[c_1c_2c_3c_4]$ va desemna expresia regulată $c_1|c_2|c_3|c_4$. Pentru intervale mai lungi şi compacte de caractere se pot indica doar capetele intervalului. Astfel `a|b|...|z` devin `[a-z]`.

Exemplu:

```
cifra → [0-9]
litera → [A-Z, a-z]
identificator → litera(litera|cifra)*
cifre → cifra+
exponent → ((E|e)(+|-)?cifre)?
fractie → (.cifre)?
numar → cifre fractie exponent
```

Observație: Un şir de relații ca cel de mai sus, în care se permite ca în expresiile regulate din membrul drept să apară *nume simbolice* definite, la rândul lor, prin alte relații anterioare, se numeşte **definiție regulată**.

Semantica

Regulile semantice ale unui limbaj precizează sensul ce se asociază construcțiilor corecte din punct de vedere sintactic. Dacă în ceea ce priveşte descrierea sintaxei notația BNF s-a impus ca o metodologie unanim acceptată, în domeniul specificării semantice coexistă o serie de metodologii şi se caută în continuare una pe deplin satisfăcătoare.

În majoritatea manualelor de utilizare şi a tratatelor de programare, semantica unui limbaj de programare se prezintă descriind efectul unei anumite construcții în limbaj natural (text însoțit de anumite desene sau scheme logice), mai mult sau mai puțin exact şi riguros. O astfel de descriere poate fi satisfăcătoare în vederea însuşirii limbajului de programare, cu atât mai mult cu cât eventualele ambiguități se clarifică prin experimentări.

Odată cu proliferarea limbajelor de programare complexe şi diverse, precum şi cu creşterea complexității, dimensiunii şi costului unor aplicații cu înalte pretenții de fiabilitate, s-a pus problema definirii în mod riguros, complet şi lipsit de ambiguități a semanticii limbajelor de programare. O astfel de definiție se va baza în mod normal pe o notație formală, matematică precis şi univoc definită. Prin aceasta ea va fi desigur mai dificil de citit necesitând o pregătire prealabilă în vederea descifrării formalismului utilizat.

Metodele de specificare formală diferă între ele prin calitatea răspunsurilor pe care le oferă la probleme de mai sus. Se pot extrage următoarele criterii de apreciere:

- **Completitudinea**: reprezintă calitatea metodei de a acoperi toate problemele legate de sintaxa şi semantica unui limbaj.
- **Simplitatea**: este măsurată prin uşurința cu care se pot realiza modele cât mai simple, indiferent de complexitatea limbajului.
- **Claritatea**: constă în înțelegerea uşoară a definițiilor; pentru aceasta metoda trebuie să permită o descriere cât mai naturală a limbajului, chiar cu riscul pierderii din conciziune.
- **Expresivitatea la erori:** este capacitatea metodei de a permite depistarea erorilor din programe.
- **Maleabilitatea:** constă în indicarea clară în definiție a locurilor în care se pot introduce restricții sau opțiuni lăsate la latitudinea implmentării; ea este legată şi de capacitatea metodei de a cuprinde toate detaliile.
- **Modificabilitatea:** reprezintă uşurința cu care se pot efectua modificări în descrierea anterioară a unui limbaj; este importantă numai în faza definirii limbajului.

Gramatici de atribute

În cazul în care procesul de traducere este coordonat de o gramatică, semantica se poate specifica prin ataşarea de *atribute semantice* la simbolurile gramaticale (neterminale şi terminale). Metoda a fost propusă de Donald Knuth în 1968 şi este foarte mult utilizată în practica realizării programelor de translatare deoarece rezolvă efectiv problemele de semantică:

- Valorile atributelor semantice se calculează prin expresii sau funcții numite *reguli semantice* asociate cu relațiile gramaticii.
- Evaluarea regulilor semantice reprezintă, de fapt, *analiza semantică*.

Regulile semantice se evaluează pe parcursul efectuării analizei sintactice, în momentul în care se aplică relația gramaticală corespunzătoare. Este şi motivul pentru care acest proces mai poartă şi denumirea de *traducere dirijată de sintaxă*.

Există mai multe modalități de asociere între regulile semantice şi relațiile gramaticale, Printre cele mai răspândite sunt *definițiile dirijate de sintaxă*.

O definiție dirijată de sintaxă este o generalizare a unei gramatici în care fiecărui simbol i se asociază o mulțime de atribute: o *gramatică atributată*. Există o mare varietate de reprezentări pentru atribute: numere, şiruri, tipuri, locații de memorie. Atributele se calculează pe măsura dezvoltării arborelui sintactic. Valoarea unui atribut la un nod din arborele sintactic se calculează printr-o regulă semantică asociată cu producția utilizată în acel nod.

Exemplu: O definiție dirijată de sintaxă pentru un program gen *calculator de birou.*

Gramatica este următoarea:

```
<linie>::=<expresie>nl
<expresie>::=<expresie>+<termen>|<termen>
<termen>::=<termen>*<factor>|<factor>
<factor>::=(<expresie>)|<cifra>
```

Definiția asociază fiecărui neterminal `<expresie>`, `<termen>` sau `<factor>` câte un atribut cu valoare întreagă numit `val`. Pentru fiecare relație corespunzătoare celor trei neterminale se calculează atributul `val` asociat neterminalului din membrul stâng în funcție de valorile lui `val` ale neterminalelor din membrul drept.

Relația gramaticală	Regulile semantice
<linie>::=<expresie>nl	tipareste(<expresie>.val)
<expresie>::=<expresie1>+<termen>	<expresie>.val::=<expresie1>.val+<termen>.val
<expresie>::=<termen>	<expresie>.val::=<termen>.val
<termen>::=<termen1>*<factor>	<termen>.val::=<termen1>.val*<factor>.val
<termen>::=<factor>	<termen>.val::=<factor>.val
<factor>::=(<expresie>)	<factor>.val::=<expresie>.val
<factor>::=cifra	<factor>val::=cifra.lexval

Fig. 2.4 Definiția dirijată de sintaxă pentru un calculator de birou simplu

Atomul cifra are un atribut `lexval` a cărui valoare este furnizată de analiza lexicală. În general, atributele simbolurilor lexicale primesc valoare în această manieră.

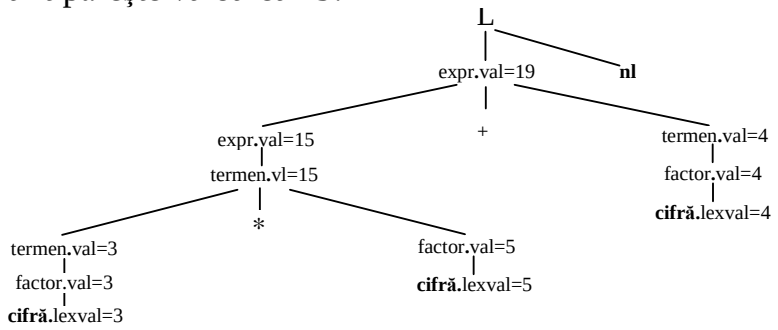
Regula semantică asociată în producția `linie::=<expresie>.nl`, pentru neterminalul de start, tipărește valoarea expresiei aritmetice generate de `<expresie>`. Se poate imagina că această regulă definește un atribut fictiv pentru neterminalul `<linie>`.

Gramaticile de atribute pot fi implementate împreună cu ambele tipuri de analiză sintactică, ascendentă și respectiv descendentă. Este metoda cea mai răspândită în prezent, în practica realizării translatoarelor.

Un arbore sintactic la care se indică, la fiecare nod, valorile atributelor, se numește **arbore sintactic adnotat** iar procesul de calcul aferent se numește **adnotarea arborelui sintactic**.

O definiție care utilizează în exclusivitate atribute sintetizate se numește definiție **S–atributată**. Un arbore sintactic pentru o definiție S–atributată poate fi întotdeauna adnotat prin evaluarea RS pentru atribute, parcurgând nodurile de jos în sus, de la frunze spre rădăcină.

Exemplu. Definiția S–atributată din exemplul anterior specifică un calculator de birou care citește o linie de intrare constând dintr-o expresie aritmetică ce conține cifre, paranteze, operatorii + și ∗, urmată de caracterul special **nl** și tipărește valoarea expresiei. De exemplu, dacă se dă expresia 3∗5+4**nl**, programul tipărește valoarea 19.



Arborele sintactic adnotat pentru 3∗5+4**nl**

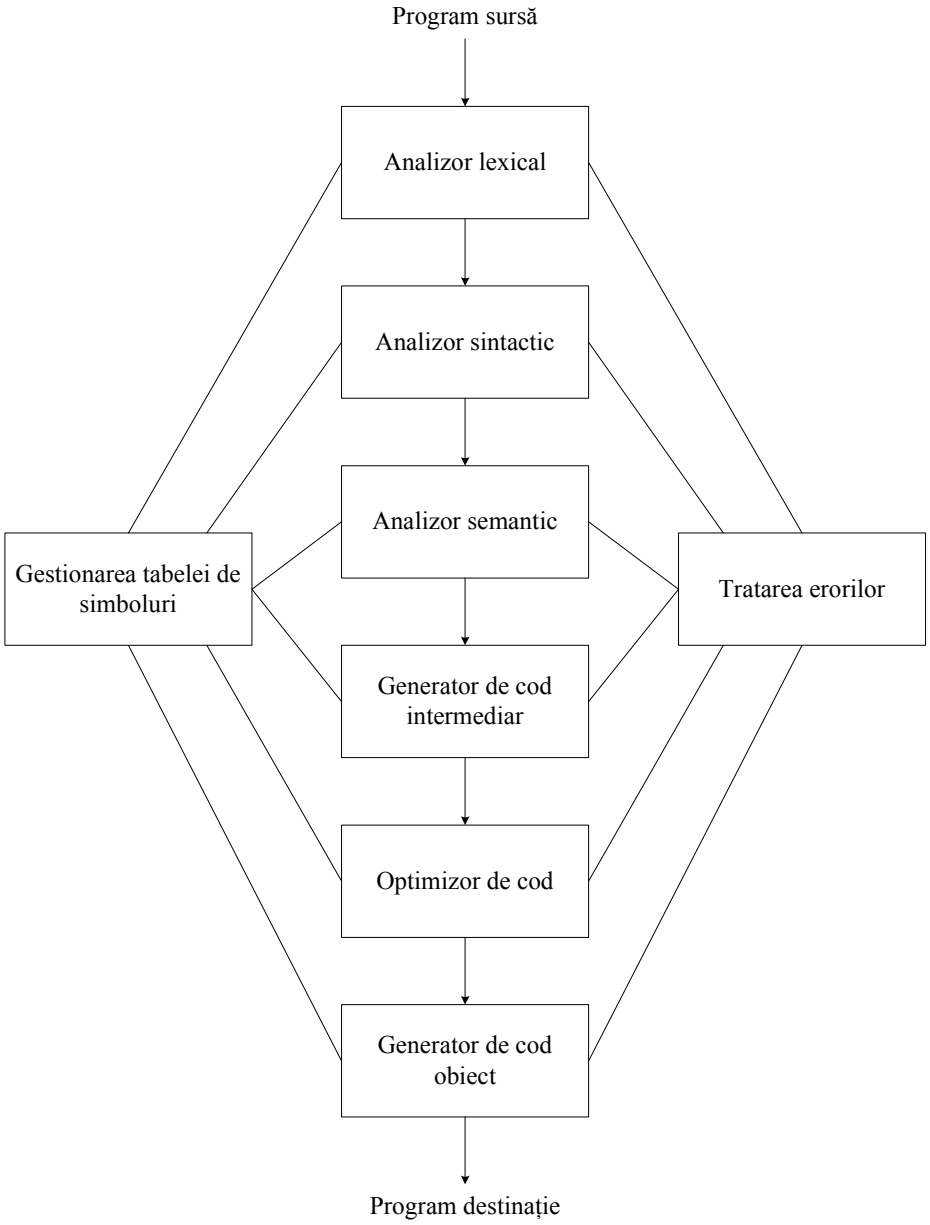
Comparație

Metoda operațională de specificare a semanticii este mai precisă decât o descriere în limbaj natural şi este foarte utilă în special pentru implementarea limbajului. Acest avantaj este şi mai evident în cazul metodei gramaticilor de atribute. O descriere completă şi lipsită de orice ambiguitate poate fi garantată doar printr-un formalism matematic adecvat. Cele două abordări matematice, cea axiomatică şi cea denotațională surprind în două ipostaze diferite sematica limbajului: prima este utilă în special în vederea verificării corectitudinii programelor; semantica denotațională se pretează cu succes în faza de proiectare a limbajului de programare, ea relevând în mod subtil o serie de aspecte privind structura internă a limbajului.

În figura se prezintă schematic o analiză comparativă a metodelor prezentate în acest capitol din perspectiva criteriilor enunțate la început.

Dintre formalismele analizate, metoda gramaticelor de atribute se apropie cel mai mult de o definiție de limbaj care să corespundă cerințelor practice privind definirea şi implementarea sa.

Criterii / Metoda	Completitudine	Simplitate	Claritate	Claritate	Expresivitate la	Maleabilitate	Modificabilitate	Aplicabilitate la implementare
Operațională	B	N	S	S	B	B	S	S
Gramatici de atribute	B	B	B	S	S	S	S	B
Axiomatică	I	S	I	B	S	S	S	N
Denotațională	S	S	S	B	N	N	B	S



Fazele unui compilator

- **Gestionarea tabelei de simboluri**

O funcție esențială a compilatorului este să înregistreze identificatorii utilizați în programul sursă şi să colecționeze informații despre diferitele atribute ale fiecărui identificator. Aceste atribute se pot referi la felul identificatorului (constantă, variabilă, funcție etc.), tipul său, memoria alocată, domeniul de valabilitate şi, în cazul numelor de funcții, informații ca numărul şi tipul argumentelor, metoda de transmitere a fiecărui argument, tipul valorii returnate.

Tabela de simboluri (TS) este o structură de date care conține o înregistrare pentru fiecare identificator, cu câmpuri pentru atributele identificatorului. Structura de date trebuie să permită regăsirea rapidă a înregistrării pentru fiecare identificator. Crearea înregistrării unui identificator în TS se face la analiza lexicală sau la cea sintactică. Informațiile corespunzătoare atributelor se introduc ulterior, pe măsură ce se avansează în procesul de compilare. Aceste informații se utilizează pentru a efectua verificările semantice (de exemplu verificările de tip) precum şi pentru generarea corectă a codului obiect.

- **Detectarea şi semnalarea erorilor**

Fiecare fază de compilare poate descoperi erori. Reacția compilatorelor la erori poate fi diferită:

a) Oprirea compilării la prima eroare, semnalarea şi corectarea ei în programul sursă, urmată de reluarea compilării de la început.
b) Tratarea erorilor detectate astfel încât compilarea să poată continua, permițând să se detecteze şi alte erori care vor fi corectate în mod global, la sfârşit. Acest mod de ”tratare” se numeşte *revenirea din eroare*.

Obiectul fazelor de analiză a programului sursă

Partea de analiză a procesului de compilare constă, la rândul ei din 3 faze:

1. Analiza lexicală sau **liniară**

Şirul de caractere formând programul sursă este citit de la stânga spre drapta şi grupat în *simboluri lexicale* care sunt secvențe de caractere având o semnificație comună. De exemplu, în instrucțiunea de atribuire `a:=b+c*10` sunt localizate următoarele familii de simboluri lexicale:

- identificatorii `a`, `b` şi `c`;
- operatorii `:=`, `+` şi `*`;
- numărul întreg `10`.

Tot pe parcursul acestei faze se elimină spațiile dintre simboluri, iar acestea sunt codificate numeric pentru a uşura prelucrarea lor în continuare.

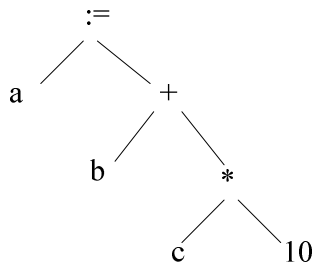
2. Analiza sintactică sau **ierarhică**

Simbolurile sunt grupate ierarhic în colecții mai mari (expresii, declarații, instrucțiuni) cu o semnificație comună, reprezentând *propozițiile gramaticale* ale programului sursă.

În timpul analizei sintactice, operațiile indicate în programul sursă sunt determinate şi înregistrate într-o structură ierarhică de arbore. Deseori se utilizează un tip special de arbore numit *arbore sintactic* în care fiecare nod reprezintă o operație iar fiii unui nod sunt argumentele operației .

Structura ierahică a unui program este, în multe cazuri, exprimată prin reguli recursive. De exemplu, sunt bunecunoscute regulile recursive de definire a expresiilor.

Similar, într-o mare categorie de limbaje, şi instrucțiunile sunt definite prin reguli recursive.



Arbore sintactic pentru `a:=b+c*10`

3. **Analiza semantică** sau **contextuală**

Efectuează verificările care țin de sensul (înțelesul) componentelor programului (constrângerile contextuale ale limbajului sursă) şi culege informații de tip pentru faza următoare: generarea de cod. Ea utilizează structura ierarhică determinată de faza de analiză sintactică pentru a identifica operatorii şi operanzii expresiilor şi instrucțiunilor.

O componentă importantă a analizei semantice este *verificarea tipurilor*. Această componentă verifică dacă fiecare operator are operanzi permişi de specificarea limbajului sursă. De exemplu, definițiile multor limbaje de programare consideră că este eroare atunci când un număr real este utilizat pentru a indexa un tablou.

O altă parte distinctă o reprezintă *analiza de domeniu* adică verificarea utilizării fiecărui identificator strict numai în domeniul său de vizibilitate.

Obiectul fazelor de sinteză a programului destinație

4. Generarea codului intermediar

După analiza sintactică şi semantică, unele compilatoare generează o reprezentare intermediară explicită a textului sursă. Se poate concepe această reprezentare intermediară ca un program pentru un calculator abstract.

Există diferite forme de reprezentări intermediare. O formă des utilizată este aşa numitul *cod cu 3 adrese* care seamănă cu limbajul de asamblare pentru un calculator în care fiecare locație de memorie poate juca rol de registru.

Codul cu 3 adrese constă dintr-o secvență de instrucțiuni, fiecare având cel mult trei operanzi. Această formă intermediară are câteva proprietăți:

a) Fiecare instrucțiune cu 3 adrese are cel mult un operator pe lângă cel de atribuire; în prealabil, compilatorul trebuie să decidă asupra ordinei în care se execută operațiile (prioritatea operatorilor);
b) Pentru a păstra valorile calculate în fiecare instrucțiune, compilatorul trebuie să genereze variabile temporare (variabile create de compilator fără corespondență directă în textul sursă);
c) Pot exista instrucțiuni cu 3 adrese cu mai puțin de 3 operanzi.

5. Optimizarea codului

Scopul acestei faze este acela de a îmbunătății codul intermediar, astfel încât să rezulte cod maşină mai rapid. În acest sens se acționează pentru eliminarea redundanțelor, a calculelor şi variabilelor inutile. Există mari diferențe între optimizarea de cod care se face în diferite compilatoare. În aşa numitele „compilatoare cu optimizare", în care acestei faze i se acordă o importanță deosebită, fracțiunea din timpul de compilare cheltuită pentru optimizare este foarte mare. Există însă şi optimizări simple care îmbunătățesc considerabil eficiența programului obiect fără a încetini prea mult compilarea.

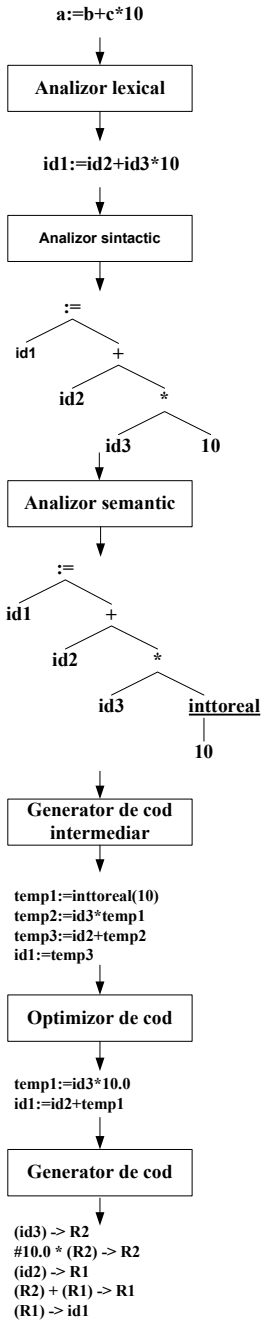
6. Generarea de cod obiect

Generarea codului destinație (obiect) este faza finală a compilatorului. Codul obiect generat poate să fie, de exemplu, cod maşină relocabil sau un cod virtual. Pe lângă transformarea instrucțiunilor intermediare în secvențe de instrucțiuni maşină (virtuale) mai trebuie rezolvate următoarele probleme:

- selecționarea şi alocarea de celule de memorie pentru variabilele din program;
- alegerea şi implementarea celor mai eficiente tehnici de acces la fiecare variabilă în parte (inclusiv la componentele variabilei) utilizând toate posibilitățile de adresare ale calculatorului: indexare, indirectare etc.;
- alocarea registrelor pentru calcule şi pentru reținerea temporară a rezultatelor intermediare.

În figura următoare se prezintă, pe faze, întregul proces de traducere al instrucțiunii *a:=b+c*10* şi principalele structuri de date asociate compilării.

Observație: Se consideră că *a, b şi c* sunt variabile reale.



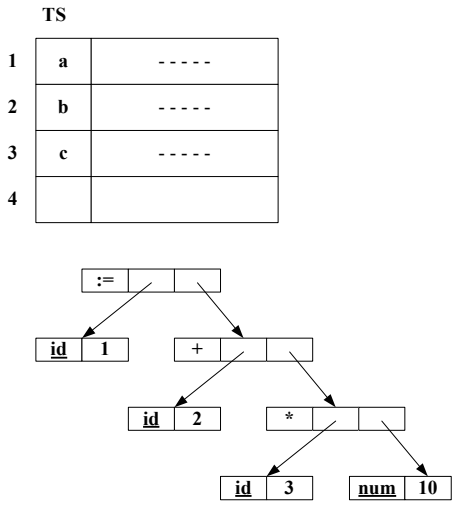


Fig. 2.9. Traducerea instrucțiunii a:=b+c*10 şi principalele structuri de date necesare